chpsim

A Simulator and Debugger for the CHP Language

—User Manual—

2002/12/18

chpsim 1.2

Marcel van der Goot
for the
Caltech Asynchronous VLSI Group

Contents

1 Language

Words in *italics* denote non-terminals or non-literal terminals; words and symbols in `typewriter font` denote keywords and literal symbols. Choices are separated by bar symbols, '[]'; braces are used for grouping ('{' is for grouping, '`{`' is a literal brace). The following subscripts are used:

- $item_{opt}$ — an optional item
- $item_{series}$ — one or more items
- $item_{list}$ — one or more items separated by commas
- $item_{seq}$ — one or more items separated by semi-colons
- $item_{tseq}$ — sequence with optional terminating semi-colon

There are also $item_{series\text{-}opt}$ etc., denoting zero or more items.

1.1 Source File

source_file:
    $required\_module_{series\text{-}opt}$ $global\_definition_{series\text{-}opt}$

global_definition:
    `export`$_{opt}$ *definition*

definition:
    type_definition
[] *const_definition*
[] *function_definition*
[] *procedure_definition*
[] *process_definition*
[] *field_definition*

The source can be distributed over multiple source files, called modules.

A definition defines the meaning of a name. If the definition is marked with `export`, the name is visible outside its own module, if that module is imported. Exporting a name is necessary if you want to refer to that name in a different module; you do not need to export objects that you do not refer to by name. E.g.,

```
export type abc = array [0..10] of pqr;
```

exports the name *abc*, but not *pqr*. It is not necessary to export *pqr* in order to access elements of an array of type *abc*.

1.1.1 Importing modules

required_module:
`requires` *string_literal*$_{list}$ `;`

The *string_literal* is a file name, denoting a module that should be imported. There is no default file name extension, but there is a search path (see `-I` in Section 2.1).

Importing a module makes the exported names of that module visible. However, if multiple imported modules export the same name, that name will not be visible. Imported names are overridden by names defined in the *source_file* itself. Importing is not recursive.

It is allowed to have circular dependencies between modules. E.g., *m*1 may require *m*2 while *m*2 requires *m*1. (It makes sense to allow this, since routines can be defined in any order; Section 1.4.6.) However, there is an important restriction on such circular dependencies:

> If module *m*1 has a circular dependency with module *m*2, then the top-level declarations of *m*1 must not depend on *m*2.

Top-level declarations are the definitions in the syntax of *source_file*, but for routines the restriction only applies to the parameters and return type, not to the body. The following example is correct code.

```
// file m1
requires "m2";

export type color = {red, green, blue};

export function f1(x: color): {0..255}
CHP
  { var y: byte;
    y := f2(0);
    f1 := y + 1
  }
```

```
// file m2
requires "m1";

export type byte = {0..255};

export function f2(x: byte): byte
CHP
  { f2 := x * 2 }

export function g2(x: byte): byte
CHP
  { g2 := f1(blue) }
```

However, the restriction prevents us from changing the return type of *f*1 to *byte*, because the return type is part of *m*1's top-level declarations. If we want to share the definition of *byte*, we should put it in a third module that is imported by both *m*1 and *m*2.

Although importing is not recursive, the notion of dependency is transitive. Hence, there can be circular dependencies involving more than two files. The restriction applies to any two modules in the same cycle.

A module should not import itself.

1.2 Types

---

type_definition:
`type` *identifier* `=` *type* `;`

type:
integer_type
[] *symbol_type*
[] *array_type*
[] *record_type*
[] *generic_type*
[] *identifier*

generic_type:
`bool` [] `int` [] `symbol`

An *identifier* used as *type* must have been defined with a *type_definition*.

As a special case, if the *type* in an exported *type_definition* is a *symbol_type* (Section 1.2.3), the symbol literals are exported as well.

The language distinguishes between generic types and specific types; each specific type can be reduced to a generic type. Typically, variables have specific types, whereas expressions have only generic types. When types need to match, such as for an assignment, it is sufficient if the generic types are equal; hence, the typing is weak. Matching of types can be verified at compile-time. However, each variable may only hold values that belong to its specific type; the simulator uses run-time checks to verify this condition. We use the word 'type' when the context makes clear whether we are referring to generic or specific types.

1.2.1 Boolean type

The generic `bool` type is an exception, in that there are no specific types. Values of `bool` type are usually produced by comparisons, but you may declare variables of type `bool` as well. The `bool` type has only two values, `true` and `false`.

1.2.2 Integer types

---

integer_type:
`{` *const_range* `}`

const_range:
const_expr `..` *const_expr*

---

integer_type describes a specific type. The two expressions denote lower and upper bounds (the lower bound must come first). The corresponding generic type is the arbitrary-precision type `int`, which is used for all integer expression evaluation. Any integer expression can be assigned to any integer variable, provided that the value lies within the bounds of the variable's specific type. Note that there is no automatic truncation of values.

It is possible to declare variables or parameters of the generic `int` type. This is not recommended for code that describes hardware, but may be useful when defining generic functions (such as *is_even*()).

1.2.2.1 Integer fields

field_definition:
`field` *identifier* `=` `[` *const_range* `]` `;`

This syntax gives a name to the specified range of bits. In this case, the higher bound may come first. Note that the field name is not tied to a particular integer type: it can be used with any integer value. See Sections 1.6.3.3 and 1.6.3.2.

1.2.3 Symbol types

symbol_type:
`{` *identifier*$_{list}$ `}`

A symbol type consists of a set of symbol literals, which are just names. For instance

```
type color = { red, orange, yellow, green, blue, purple };
```

Unlike enumeration types in some languages, there is no ordering among symbol literals, nor is there an implied mapping to integers. The generic type for all symbol types is `symbol`. Since all symbol values belong to the generic type, they are identified only by their name: if types *color* and *fruit* both have an *orange* value, these values are identical. The first occurrence of a name in a *symbol_type* declares the name as a symbol literal; subsequent occurrences (in the same scope) do not count as declarations. Within a single *symbol_type*, each name should occur only once.

To export symbol literals, the *symbol_type* must be given a name with an exported *type_definition* (Section 1.2).

1.2.4 Array types

array_type:
`array` `[` *const_range*$_{list}$ `]` `of` *type*

Having multiple ranges is merely a short-hand for nested array types: the two types

```
array [1..10, 0..5] of byte
array [1..10] of array [0..5] of byte
```

describe exactly the same specific type. Internally, the simulator always uses the latter form.

The generic type of a specific array type is obtained by omitting the bounds, and replacing the element type by its generic type. For example,

```
var a: array [1..10] of byte;
var b: array [0..5] of {-100..100};
var c: array [1..10] of {red, green, blue};
```

Arrays *a* and *b* have the same generic type, namely `array of int`, but *c* does not. When an array value is assigned to an array variable, the number of elements must be equal, and the assignment of each individual element must be valid. Hence, $a := b$ can never be correct, but $b := a[i..j]$ may be correct.

1.2.5 Record types

record_type:
`record` `{` *record_field*$_{tseq}$ `}`

record_field:
identifier$_{list}$: *type*

The field names are local to each specific *record_type*. The generic type of a specific record type is obtained by omitting the field names, and replacing each field type by its generic type. For example,

```
var a: record { x, y: byte };
var b: record { p, q: {-100..100} };
var c: record { x, y, z: byte };
```

Both *a* and *b* have the same generic type, namely

```
record { int; int }
```

but *c* does not. Assignment of record types requires that each of the field assignments is valid.

The field names are always accessed through an object of the record type; hence, they need not be exported.

1.3 Constants

---

const_definition:
`const` *identifier* {`:` *type*}$_{opt}$ *initializer* `;`

initializer:
`=` *const_expr*

If the *type* is specified, the *initializer* must be a value of that type; otherwise, the (generic) type of the constant is determined directly from the initializer.

1.4 Routines

The term ‘routine’ refers to functions, procedures, and processes.

1.4.1 Parameter passing

---

value_parameter:
`val`$_{opt}$ *identifier*$_{list}$: *type*

result_parameter:
`res` *identifier*$_{list}$: *type*
[] `valres` *identifier*$_{list}$: *type*

The term ‘parameter’ refers to formal parameters; ‘argument’ refers to actual parameters.

The parameter passing mechanism for functions and procedures is value-result (a.k.a. copy-restore) passing: `val` and `valres` parameters get their initial value assigned from the corresponding arguments at the very beginning of the call. At the very end of the call, the final values of `res` and `valres` parameters are copied to the corresponding arguments of the call. In a hardware implementation, this mechanism easily translates to receiving and sending initial and final values. During the call, the parameters are local variables of the routine (note that `res` parameters are not initialized).

```
procedure g(val p: int; valres q: int; res r: int)
CHP { q := q + p;
      p := q;
      r := p + 1;
    }
```

Suppose we have $x = 3$ and $i = 1$, then call $g(x, i, a[i])$. Afterward $x = 3$, $i = 4$, and $a[1] = 5$. Note that the location of $a[i]$ was determined before the call, so that $a[1]$ is modified rather than $a[4]$.

The type requirements for parameter passing are the same as for assignment. In addition, the arguments for `res` and `valres` parameters must be ℓ-values (writable locations). It is an error to pass the same location for two result parameters in the same call. E.g., with the above example, $g(x, i, i)$ would be wrong.

1.4.2 Functions

---

function_definition:
`function` *identifier* `(` *value_parameter*$_{seq}$ `)` `:` *type* *chp_body*

A function has a return type and is called as part of an expression. The parameters of functions are always `val` parameters. Inside the function, the function name acts like a `res` parameter. The value you assign to it corresponds to the function's return value.

A function may contain nested functions and procedures, but no processes.

Because there are no global variables, and because function parameters are value parameters, functions are free of side-effects. Also, functions do not have a persistent state.

1.4.3 Procedures

---

procedure_definition:
`procedure` *identifier* `(`
{*value_parameter* `[]` *result_parameter*}$_{seq\text{-}opt}$ `)` *chp_body*

A procedure has no return type; its call is a statement.

A procedure may contain nested functions and procedures, but no processes.

A procedure can modify its environment by modifying result parameters, but otherwise has no side-effects, nor a persistent state.

1.4.4 Processes

---

process_definition:
`process` *identifier* `(` *meta_parameter*$_{seq\text{-}opt}$ `)`
`(` *port_parameter*$_{seq\text{-}opt}$ `)` *process_body*

port_parameter:
{*identifier* *direction*}$_{list}$ `:` *type*
`[]` *identifier*$_{list}$

direction:
`?` `[]` `!`

process_body:
    chp_body
[] *meta_body*

Processes have no value or result parameters; instead they have meta parameters and ports. Port parameters of the first form are data ports, used for sending and receiving values. A '?' indicates an input port, a '!' indicates an output port. The *type* is the data type of the port.

Port parameters of the second form are synchronization ports.

Processes are not called, but instantiated. Within a process, the meta parameters act as constants (e.g., they can be used in type definitions).

There are two types of processes, meta processes and CHP processes. Meta processes serve to instantiate other processes, eventually resulting in a process graph with only CHP processes. Meta processes cannot contain communications. Only when the complete process graph has been created, does the execution of the CHP program start. Meta processes are explained in Section 1.7.

1.4.5 Routine body

---

chp_body:
    `chp { {`*definition* `[]` *declaration*`}`$_{\textit{series-opt}}$
        parallel_statement$_{\textit{tseq-opt}}$ `}`

declaration:
    `var` *identifier*$_{\textit{list}}$ `:` *type* *initializer*$_{\textit{opt}}$ `;`

Definitions inside a routine body are local to that routine; unlike global definitions, they cannot be exported. Variables are always local to a routine.

If a variable has an initial value, the value must be of the variable's type. Unlike for constants, a variable's type cannot be omitted.

1.4.6 Scope and order of definition

All names, including symbol literals, are in the same name space. However, there are multiple nested scopes, due to the nesting of definitions. Scope is always static, i.e., the meaning of a name can be determined at compile time. The outermost scope level contains the names exported by imported modules. The next scope level is the module (*source_file*) itself. Definitions and declarations inside a body are local to that body. Parameters of routines are also local to that routine's body.

A name can always be redefined in a nested scope, hiding the original meaning of the name. Each name may be defined or declared only once in a particular scope, except for the names exported by imported modules. In case of the latter, if different modules export the same name, neither meaning of the name is visible. As explained in Section 1.2.3, all symbol literals belong to the same generic type. Symbol literals are only declared the first time they are encountered (in a scope).

Most names are visible in their own scope, and in nested scopes if they have not been redefined. However, to avoid shared variables, variables must always be local to the routine that uses them.

Routines can be defined in any order, but all other names must be defined or declared before they are used. (However, if a process definition and instantiation occur at the same scope level, then the definition must precede the instantiation.)

1.5 Statements

---

parallel_statement:
    statement $_{list}$

statement:
    `skip`
[] *assignment*
[] *communication*
[] *loop_statement*
[] *selection_statement*
[] *procedure_call*
[] `{` *parallel_statement* $_{tseq}$ `}`

Statements separated by commas are executed in parallel. A variable that is only read may be read by multiple statements in parallel. However, a variable that is modified may be accessed by only one of a group of parallel statements. With respect to this rule, communications count as modification of the port. If this exclusion rule is violated, the effect is undefined.

Statements separated by semi-colons are executed in sequence. From the syntax it follows that commas bind tighter than semi-colons. Braces can be used to alter the binding.

A `skip` statement has no effect.

1.5.1 Assignment

---

assignment:
    lvalue `:=` *expr*
[] *lvalue* `+`
[] *lvalue* `-`

lvalue:
    postfix_expr

An expression is an ℓ-value if it corresponds to (part of) a variable. In Section 1.6 we indicate which expressions are ℓ-values.

An assignment with an `:=` symbol requires that both expressions have the same generic type, and that the value belongs to the type of the assigned variable.

The other two forms of assignment require that the ℓ-value has type `bool`; a `+` sets the boolean to `true`, a `-` sets it to `false`.

1.5.2 Communication

---

communication:
    port_expr
[] *port_expr* `!` *expr*
[] *port_expr* `?` *lvalue*
[] *port_expr* `!` *port_expr* `?`
[] *port_expr* `#?` *lvalue*

port_expr:
expr

The different forms of *communication* are referred to as sync, send, receive, pass, and peek, respectively.

The *port_expr* must be a port. For a sync, the port must be a synchronization port; for all other communications it must be a data port.

For a send, the port must be an output port, and the expression must be valid for the data type of the port.

For a receive and a peek, the port must be an input port, and the value received is assigned to the ℓ-value. The peek receives a value without removing it from the port; hence, a subsequent receive or peek will receive the same value.

The first port of a pass must be an output port, the second must be an input port. The pass receives a value from the input port, and simultaneously (without introducing slack) sends it via the output port. The value received must be valid for the data type of the output port.

All communication actions, including the peek, suspend until they can complete. A suspended peek does not cause the probe of the connected port to be true. A suspended pass passes the probe from one side to the other, but does not by itself cause a true probe. The other communications, when suspended, cause the probe of the connected port to be true.

1.5.3 Repetition

---

loop_statement:
`*[` *guarded_command* { `[]` *guarded_command* }*series-opt* `]`
`[]` `*[` *guarded_command* { `[:]` *guarded_command* }*series-opt* `]`
`[]` `*[` *parallel_statement*$_{tseq}$ `]`

guarded_command:
bool_expr `->` *parallel_statement*$_{tseq}$

bool_expr:
expr

The guard of a guarded command, i.e., the expression before the arrow, must have `bool` type. Execution of a guarded command consists of execution of the statement sequence.

The first two forms of the *loop_statement* choose one guarded command that has a true guard, and execute it. This process is repeated until no guard is true. If the guarded commands are separated by `[]` symbols, it is an error if more than one guard is true (at the time the choice is made). If the separator is `[:]`, an arbitrary choice is made among the guarded commands with a true guard.

The last form of the *loop_statement* executes the statement sequence repeatedly, forever.

1.5.4 Selection

---

selection_statement:
`[` *guarded_command* { `[]` *guarded_command*}*series-opt* `]`
`[]` `[` *guarded_command* { `[:]` *guarded_command*}*series-opt* `]`
`[]` `[` *bool_expr* `]`

The first two forms of the *selection_statement* wait until one of the guarded commands has a true guard, then execute one of the guarded commands that has a true guard. The distinction between the first two forms is the same as for the *loop_statement*: with the `[]` separator it is an error if more than one guard is true.

The last form of selection statement simply waits until the expression (which must have `bool` type) is true.

1.5.5 Procedure call

---

procedure_call:
identifier { (*expr* *list-opt*) }*opt*

The *identifier* must be the name of a procedure (not of a function). The argument expressions match the parameters of the procedure, in order. Let expression *x* match parameter *p*. If *p* is a `val` or `valres` parameter, the assignment *p* := *x* is performed at the start of the procedure call. If *p* is a `res` or `valres` parameter, the assignment *x* := *p* is performed at the end of the procedure call. In all cases, the standard rules for the assignment apply. In addition, the same ℓ-value may not be used for two different result parameters.

The parentheses are optional if the argument list is empty.

1.6 Expressions

---

const_expr:
expr

expr:
binary_expr

An expression is a constant expression if all its constituent expressions are constants.

1.6.1 Binary expressions

---

binary_expr:
prefix_expr
[] *binary_expr* *binary_operator* *binary_expr*

binary_operator:
^
[] * [] / [] % [] mod
[] + [] - [] xor
[] < [] <= [] > [] >=
[] = [] !=
[] & [] |
[] ++

The ambiguity in the syntax of binary expressions is resolved by using operator precedence. Each line in the definition of *binary_operator* corresponds to a precedence level: the first line has the highest level, i.e., '^' binds the tightest. Operators listed on the same line have equal precedence. Among operators with equal precedence the order is left-to-right, i.e., all operators are left-associative.

A binary expression is not an ℓ-value.

1.6.1.1 Arithmetic operators

The following operators require that their operands have generic type `int`; the result of the operation has type `int`:

```
^  *  /  %  mod  +  -
```

'`^`' denotes exponentiation: `2^N` means 2^N. Since the result is an integer, the second operand must be ≥ 0 ($x^0 = 1$ always).

'`/`' is integer division (rounding towards 0). Consider a/b. If a and b have the same sign, the result is ≥ 0; otherwise the result is ≤ 0. '`%`' is the remainder of division. The sign of $a\%b$ is the sign of a.

`mod`, on the other hand, is the standard modulo operation, which always yields a non-negative result (a `mod` $b = a$ `mod` $|b|$). The operations are summarized by the following example.

$10/\ 3 = 3$	$10\%\ 3 = 1$	10 `mod` $3 = 1$
$-10/\ 3 = -3$	$-10\%\ 3 = -1$	-10 `mod` $3 = 2$
$10/-3 = -3$	$10\%-3 = 1$	10 `mod` $-3 = 1$
$-10/-3 = 3$	$-10\%-3 = -1$	-10 `mod` $-3 = 2$

The `/`, `%`, and `mod` operators require that their second operand is not 0.

1.6.1.2 Comparison

The `<`, `<=`, `>`, and `>=` operators all yield a `bool` result. Their operands must either both have generic type `int`, or both have type `bool` (where `false<true`).

The `=` and `!=` operators also yield a `bool` result. They require that both operands have the same generic type.

1.6.1.3 Logical and bitwise operators

The `&`, `|`, and `xor` operators require that either both operands have type `bool`, or both operands have generic type `int`.

If the operands have type `bool`, the result also has type `bool`.

If the operands have generic type `int`, the result also has type `int`; in this case, the operation is a bitwise operation. Note that precedence of the operators is what is usually expected for the logical operations. Due to the arbitrary precision of integer expressions, care must be taken when bitwise operations are applied; see Section 1.6.2.2.

1.6.1.4 Concatenation

The `++` operator denotes array concatenation. It requires that both operands have the same generic array type; the type of the result is that generic array type.

1.6.2 Prefix expressions

---

prefix_expr:
    postfix_expr
[] # { *port_expr*$_{list}$: *bool_expr* }
[] *prefix_operator* *prefix_expr*

prefix_operator:
`+` `[]` `-` `[]` `~` `[]` `#`

A prefix expression is not an ℓ-value.

1.6.2.1 Arithmetic operators

The `+` and `-` operators require that their operand has generic type `int`, which is also the result type.

1.6.2.2 Logical and bitwise operator

The `~` operator can be applied to a `bool` operand, in which case it denotes negation and yields a `bool` result. It may also be applied to an operand with generic type `int`, in which case it is applied bitwise, resulting in the one's complement operation (the result type is `int`).

Because `int` has arbitrary precision, care must be taken when bitwise operations are applied. For the purpose of bitwise operations, an integer is considered as an infinite array of 0s and 1s, using the standard 2's complement notation. Consequently, there is always an infinite number of equal sign bits. This means that the `~` operator always changes the sign of an integer. While this is sound (since, by definition, $-x = \sim x + 1$), it may be unexpected if you intended a number to be an unsigned integer. For example, `~0 = -1`.

The binary bitwise operators likewise require care with the sign bits. Consider

```
var x: {-128..127}
```

In this case, bit 7 of x is the sign bit. However, if $x > 0$, then `x|2^7` is larger than 128, not negative, because the infinite sequence of 0 bits has not been changed. To change the sign, you would need to do `x|-2^7`.

1.6.2.3 Probe

The operand of the probe ('#') prefix operator must be a port. The expression returns a `bool` result, `true` if a subsequent communication action on the port will complete.

The value probe, the form of *prefix_expr* with braces, takes a list of ports. The *bool_expr* that follows can refer to the input ports in the list as if they were variables: each input port stands for the value that a subsequent communication will receive. The value probe is true if the individual probes of the each of the ports is true, and the expression evaluates to true.

Neither form of probe suspends.

1.6.3 Postfix expressions

---

postfix_expr:
atom
`[]` *array_access*
`[]` *record_access*

1.6.3.1 Indexing of arrays

---

array_access:
postfix_expr `[` *expr* *list* `]`
`[]` *postfix_expr* `[` *expr* `..` *expr* `]`

In both cases the *postfix_expr* must have a specific array type, or have generic type `int` (see Section 1.6.3.2). The whole expression is an ℓ-value if the *postfix_expr* is an ℓ-value.

The expression $x[3, 4, 5]$ is exactly the same as $x[3][4][5]$; internally, the simulator uses the latter form.

If the *postfix_expr* has a specific array type with element type T, then the first form of *array_access* has result type T. The second form (sometimes called a *slice*) has as result type the generic type of the *postfix_expr*. Note that you can only index an array with a specific type, i.e., an array with known bounds. Hence $x[1..4][2]$ is not a valid expression. The indices must be within the bounds of the array. For a slice, the smallest index must come first.

1.6.3.2 Indexing of integers

The same syntax used to index arrays can be used to access bits of an integer. In this case, the indices must be ≥ 0. The first form, with a single index, has result type `bool`. The second form (the slice) has result type `int`.

As explained in Section 1.6.2.2, some care is necessary when treating integers as arrays of bits. In particular, you should never set the sign bit directly. E.g.,

```
var x: {-128..127};
x[7] := true
```

If x was > 0, then the assignment makes $x > 128$, not negative, because the infinite sequence of 0 sign bits has not been changed. The following procedure sets the sign bit correctly.

```
// set sign bit x[pos] to b
procedure set_sign(valres x: int; pos: int; b: bool)
CHP
  { [   b -> x := x | -2^pos
    [] ~b -> x := x & (2^pos - 1)
    ]
  }
```

The value of a slice is obtained by treating the specified bits as an unsigned integer. Hence, with the above x, $x := x[0..7]$ is unsafe, because the result has range $0..255$. When taking a slice of an integer, it is allowed to put the largest index first.

1.6.3.3 Accessing fields of records

---

record_access:
postfix_expr `.` *identifier*

The *postfix_expr* must have a specific record type or have generic type `int`. The expression is an ℓ-value if the *postfix_expr* is an ℓ-value.

In case of a record type, the identifier must be one of the record's field names. The result type is the type of the field.

If the *postfix_expr* is an integer, the identifier must be a field name defined with a *field_definition* (Section 1.2.2.1). This specifies a slice of the integer, and is completely equivalent to the array notation for slices explained in Section 1.6.3.2.

1.6.4 Atoms

atom:
identifier
[] *literal*
[] *array_constructor*
[] *record_constructor*
[] *function_call*
[] (*expr*)

An identifier used as *atom* is an ℓ-value if it is the name of a variable or parameter; the name of a function is also an ℓ-value inside that function. (As mentioned in Section 1.4.6, variables, parameters, and function names used as variables, can only be accessed in their own scope, not in nested routines.) Other identifiers used as *atom* should be the names of constants or symbol literals. Parentheses do not affect whether an expression is an ℓ-value.

1.6.4.1 Array constructor

array_constructor:
[$expr_{list}$]

The expression must all have the same generic type T. The result type is a generic `array of` T. An *array_constructor* is not an ℓ-value.

1.6.4.2 Record constructor

record_constructor:
{ $expr_{list}$ }

The result type is a generic record with as field types the generic types of the expressions. A *record_constructor* is not an ℓ-value.

1.6.4.3 Function call

function_call:
identifier ($expr_{list}$)

The identifier must be the name of a function. The requirements for the arguments are the same as for procedure calls (Section 1.5.5), but only value parameters are allowed. Note that a function must have at least one parameter. The type of the expression is the return type of the function.

A function call is not an ℓ-value. If all arguments are constants, the function call is also a constant.

1.6.5 Literals

literal:
integer_literal
[] *character_literal*
[] *string_literal*
[] *symbol_literal*
[] *boolean_literal*

symbol_literal:
identifier

boolean_literal:
`false` [] `true`

Literals are never ℓ-values. Integer literals have type `int`. Character literals have type

```
type char = {0..127}
```

Their values correspond to ASCII codes. A string literal is a 0-terminated generic `array of int`, similar to an *array_constructor*.

A *symbol_literal* must have been declared as part of a *symbol_type*.

1.7 Meta processes

---

meta_body:
`meta { {`*definition* [] *meta_declaration*`}`$_{\textit{series-opt}}$
parallel_statement$^*_{\textit{tseq-opt}}$ `}`

A meta process is a process with a *meta_body*. The form of a meta process is nearly the same as that of a CHP process, but there are some statements and declarations that can only occur in one type of process and not in the other. The main difference is with respect to execution: The simulator starts by executing meta processes, which have as goal to create the graph of CHP processes; this is the instantiation phase. If meta process *P* instantiates process *Q* (which may be a meta process itself), then *Q* does not start executing until *P* has terminated. If *Q* is a CHP process, then *Q* does not start executing until all meta processes have terminated.

The instantiation phase ends when all meta processes have terminated. Once that happens, the execution phase starts by executing all instantiated CHP processes in parallel. The CHP processes model the actual hardware; the meta processes just serve to describe the hardware configuration.

The '*parallel_statement**' in a *meta_body* has the same form as a regular *parallel_statement*, except that '*statement*' excludes communications and is extended as follows.

---

statement:
. . .
[] *meta_binding*
[] *connection*

Furthermore, a meta process may not contain any probes.

1.7.1 Process instances

---

meta_declaration:
`instance` *identifier*$_{\textit{list}}$ `:` *process_type* `;`
[] *declaration*

process_type:
identifier
[] `array [` *const_range*$_{\textit{list}}$ `] of` *process_type*

An `instance` declaration instantiates one or more processes. The *identifier* in *process_type* must be the name of a process (as defined with `process`); this may be a meta process or a CHP process. Normally identifiers are declared immediately when they are encountered, but as a special case, instance names are only declared after parsing the *process_type*: this means the instance name may be identical to the process name (assuming they are declared at different scope levels).

Although an `instance` declaration creates the process instance, it does not specify the values of the instance's meta parameters, nor its connections to other processes. These must be specified with subsequent statements.

1.7.2 Meta parameters

---

meta_parameter:
    identifier $_{list}$ `:` *meta_type*

meta_type:
    type

Meta parameters get their value during the instantiation phase. Although their values are not known at compile-time, nevertheless they are considered constants. In the future there may be meta types that are not CHP types.

Meta parameters are given a value with a *meta_binding*.

---

meta_binding:
    instance_expr `(` *expr* $_{list}$ `)`

instance_expr:
    postfix_expr

The *instance_expr* must be one of the process instances declared with `instance`. The argument expressions must match the meta parameters of the corresponding process, just as with function parameters.

There must be a *meta_binding* for each process instance that has meta parameters.

1.7.3 Connecting processes

---

connection:
    `connect` *connection_point* `,` *connection_point*

connection_point:
    port_expr
`[]` *instance_expr* `.` *identifier*

The first form of *connection_point* identifies a port of the current process. The second form identifies a port of an instantiated process.

For data ports, if both *connection_point*s have the same form, they must be ports with opposite directions (one input, one output). However, if one *connection_point* has the first form and the other the second form, both must have the same direction. The data types of connected ports must be compatible.

Using a port of the current process indicates a pass-through: the port is not used for direct communication, but instead is merely a 'wire' between two other ports. The current implementation does not perform (run-time) range checks on the data type of a pass-through port, only on ports that are actually used in communication actions.

1.8 Lexical tokens

The description of lexical tokens is slightly informal. A token may not contain white space, unless explicitly allowed. In some cases tokens must be separated by white space to avoid ambiguity.

1.8.1 Comments

---

comment:
`/*` *any*$_{\textit{series-opt}}$ `*/`
[] `//` *any-except-linebreak*$_{\textit{series-opt}}$ *linebreak*

A comment of the first form cannot contain '`*/`'.

1.8.2 Integers

---

integer_literal:
decimal
[] {`0x` [] `0X`} {*hex_digit* [] `_`}$_{\textit{series}}$
[] {`0b` [] `0B`} {*binary_digit* [] `_`}$_{\textit{series}}$
[] *decimal* `#` {*based_digit* [] `_`}$_{\textit{series}}$

decimal:
decimal_digit {*decimal_digit* [] `_`}$_{\textit{series-opt}}$

Digits > 9 are `a..z` (case-insensitive). Integers can contain underscores, but not as first character. The '#' notation allows for integers in any base, $1 < \textit{base} \leq 26$.

1.8.3 Identifiers

---

identifier:
{*letter* [] `_`} {*letter* [] *digit* [] `_`}$_{\textit{series-opt}}$

Identifiers may not be keywords; they are case-sensitive.

1.8.4 Keywords

Keywords are case-insensitive.

1.8.5 Characters and strings

---

character_literal:
`'` {*char_character* [] `\` *printable_character*} `'`

char_character:
printable_character *except* `\`

A backslash is used as escape character. The following escapes are recognized:

`\a`	BELL	`0x7`
`\b`	BS	`0x8`
`\t`	TAB	`0x9`
`\n`	LF	`0xA`
`\v`	VT	`0xB`
`\f`	FF	`0xC`

`\r`	CR	`0xD`
`\q`	XON	`0x11`
`\s`	XOFF	`0x13`
`\"`	"	`0x22`
`\'`	'	`0x27`
`\\`	\	`0x5C`

A *printable_character* is any character in the ASCII range `0x20` through `0x7E`, i.e., a space character or a character that involves ink when printing.

---

string_literal:
" {*string_character* [] \ *printable_character*}*series* "

string_character:
printable_character *except* " *and* \

Escapes for strings are the same as for characters.

2 Simulation

2.1 Command line arguments

The `chpsim` program takes the following command line arguments.

source_file The top-level module. Only one source file is specified on the command line; other modules are read based on `requires` clauses.

`-main` *process*
The execution starts with one instance of this process. This must be a process without meta parameters and port parameters. If no `-main` option is specified, the initial process is *main*.

`-I` *directory*
Appends the directory to the module search path. If module *A* has a `requires "B"` clause, the simulator first looks for *B* in the directory where it found *A*. If *B* is not found in that directory, the simulator checks each of the directories in the search path, in order. The search path does not apply to the top-level module, nor does it apply if *B* is an absolute path name or starts with '`.`' or '`..`'. See also the `-v` option.

`-I-` Clears the search path.

`-batch` By default an interactive session is started, as described in Section 2.2. With this option, the simulator executes the whole program without stopping for user commands.

`-log` *file* A copy of all user commands and simulator output is written to this file. When `-batch` is used, the default log file is *stderr*. (The log file only applies to simulator output, not to output generated by the CHP program; see the `-stdout` option.)

`-stdout` *file*, `-o` *file*
Specifies the file to use for the *print* and *show* procedures described in Section 2.3, and for *stdout* defined in Section 2.4. The default is the standard output.

`-v` Prints version information, the search path, and the name of each file that is read.

`-trace` *instance*
Trace this instance, in the same way as the `trace` command of Section 2.2.4. This is useful with the `-batch` option (you probably also want to use `-log`). Instance names are described in Section 2.2; they always start with a '/'.

`-traceall`
Trace all process instances. This will generate a lot of output for anything but the shortest programs. In most cases, judiciously placed breakpoints are more helpful.

2.2 Execution

At any time, the simulator executes a number of threads in parallel. Typically, there is one thread per process instance, but if a process executes a parallel statement (statements separated by commas), it has temporarily multiple threads. Each thread can be active (ready) or suspended. The simulator's `print` command lists the current threads:

```
(cmnd?) print
(active) /q[0]/r at example.chp[42:2]
(active) /tgt at example.chp[53:17]
(suspended) /src at example.chp[63:17]
```

The example shows that there are three threads, corresponding to three process instances. The names starting with a slash are hierarchical instance names, constructed from the identifiers in instance declarations (Section 1.7.1). The initial process has as name merely a slash. Hence, `/q[0]/r` is instance `r` created by `/q[0]`, which itself was instantiated by the initial process. An indication like `example.chp[42:2]` gives, respectively, the source file, the line number, and the position on the line, of the statement that will be executed next by the thread.

The simulator executes one (non-composite) CHP statement at a time, chosen from among the active threads. Execution is fair: an active statement will eventually be executed; a suspended statement will eventually be checked to determine whether it can be made active again.

The simulation consists of two phases, the instantiation phase and the execution phase. The simulation starts by creating a single instance of the initial process. Typically, this is a meta process that instantiates other (child) processes. As explained in Section 1.7, child processes only start executing when their parent process has terminated. Furthermore, CHP processes only start executing when all meta processes have terminated, which is the end of the instantiation phase. (The initial process may be a CHP process, in which case the execution phase starts immediately.)

2.2.1 The current statement

When the simulator stops to allow input of a command, it prints the statement that will be executed next and the instance it belongs too. (Except after a warning or error, in which case the printed statement was already executed.) This is the current statement, which is the focus of most simulator commands. When no process instance is mentioned, most commands apply to the current statement or the current instance. Some commands allow you to select an instance explicitly. For convenience, several of the simulator commands (such as `trace`) allow you to reference process instances that have not yet been created.

The `view`, `up`, and `down` commands let you change the instance and statement that have the focus. This is mainly useful when you want to print variable values (because variables must be in scope), but also changes the default focus for the other simulator commands. However, these commands never affect the actual order of execution: the next statement is the original 'current statement.' See Section 2.2.5 for more details.

Commands may be abbreviated, usually to a single letter.

2.2.2 Steps

The simulator repeatedly executes statements, or steps, until there is a reason to ask for command input. The simulator stops at the beginning of the instantiation phase, and at the beginning of the execution phase. When the simulator is stopped, you can enter commands to inspect variables, to set breakpoints, etc.. To make the simulation continue, you must enter one of the three commands `step`, `next`, or `continue`. An empty command is equivalent to one of these three, depending

on what made the simulator stop: if the simulator stopped because of a `step` or `next` command, that is also the default; in all other cases the default is `continue`.

The `step` command tells the simulator to execute one step, i.e., one statement, of the current process. This is the statement printed before the command prompt. For instance:

```
(step) /src at example.chp[63:17]
        p!n
(cmnd?)
```

The current process is `/src`, which is about to execute `p!n`. Entering `step` (or, in this case, an empty command) allows execution of `p!n`, and forces a stop at the next statement of `/src`. The `step` command refers to the current process only: between the current step and the next stop, there may be multiple execution steps of other processes.

The `next` command is nearly the same as the `step` command, but it treats function and procedure calls as a single step. I.e., if the current statement is a procedure call, `next` will stop at the statement after the call, whereas `step` will stop at the first statement of the called procedure.

The `step` and `next` commands can take an instance name as argument. In that case the simulator will stop when it reaches the next statement of that process instance.

The `continue` command continues execution until there is reason to stop again, such as a breakpoint.

2.2.2.1 Execution of functions

Since expressions are always part of a statement, execution of a statement as a single step implies that expression evaluation is atomic. However, this poses a problem for function evaluation: a function call is part of an expression, hence atomic, yet its evaluation consists of multiple statements. The simulator solves this by executing the function in isolation: while executing the function, it ignores all other threads until the function has finished. Since functions do not interact with other threads, their atomicity is actually a non-issue; it is mentioned only because it can be observed in the simulator.

Function calls with constant arguments are themselves constant (Section 1.6.4.3). You may observe that such a call is indeed executed only once.

Procedure calls are statements, and are therefore treated like other composite statements: they are not atomic.

2.2.3 Breakpoints

The `break` command sets a breakpoint. Whenever execution reaches a breakpoint, the simulator stops to allow command input. Breakpoints are associated with a statement in the code, not with a particular instance.

By itself, `break` sets a breakpoint at the current statement.

The following form of `break` sets a breakpoint at the statement at or near the indicated position.

```
break "filename" linenumber : position
```

Only the line number is required. If a file name is specified, it must be between quotes. If a position on the line is specified, it must be preceded by a colon.

This third form of `break` sets a breakpoint at the beginning of the specified routine (a process, procedure, or function).

```
break "filename" routine
```

Again, the file name is optional. If the routine is not a top-level routine, you can use a dot to construct a hierarchical name: use `break g.f` to stop at function `f` which is declared inside procedure `g`. A file name need only be specified if otherwise the top-level routine name is not unique.

To remove a breakpoint, use the `clear` command when stopped at the breakpoint. There is no command to remove a breakpoint that the simulator is not stopped at.

2.2.4 Tracing

When a process instance is traced, the simulator prints every statement executed by that instance, without stopping (unless there is another reason to stop).

`trace` by itself starts tracing the current process instance. If an instance name is specified, that instance will be traced.

To stop tracing, use `clear trace`, optionally followed by an instance name. `clear` followed by an instance name stops tracing and also cancels a pending `step` or `next` for that process (i.e., execution will not stop when the next statement of that process is reached).

2.2.5 Inspecting the state

The `where` command prints the current call stack, i.e., the sequence of nested procedure and function calls that led to the current statement. You can move the focus up or down this stack with the `up` and `down` commands (both take an optional number of steps as argument). The 'up' direction is towards the caller. Moving the focus is useful when you want to print variables, because only variables of the current routine are visible.

You can shift the focus to a different process with the `view` command. (Without argument this simply prints the current statement.) Again, this is useful if you want to print variables of that process. As mentioned before, changing the focus changes the default process for commands like `step`; however, it does not change the actual scheduling of statements.

The `print` command by itself shows all existing threads.

`print` followed by an instance name prints the values of the instance's meta parameters, the other processes the instance is connected to, and the current position in the code.

`print` followed by a variable or constant name prints the value of that variable or constant. The name must be in scope at the current position in the code.

2.2.6 Other commands

The `help` command prints a command summary.

The `batch` command switches to non-interactive execution, without stopping for more user input, just like the `-batch` option described in Section 2.1.

Interrupting the program with `ctrl-C` forces the simulator to stop at the next statement, regardless of the instance. This is useful if the program gets stuck in an infinite or very long loop. (In batch mode, `ctrl-C` simply terminates the simulation.)

The simulator also stops immediately after a statement that caused a warning or error message, so that you can inspect variables etc.. The computation cannot continue following an error; instead, the simulator will terminate.

The `quit` command also terminates the simulation.

2.3 Built-in procedures

The simulator has a few built-in procedures that can be called from your CHP program to help with testing and debugging. These are always available, unless you have redefined the identifiers.

```
procedure step()
```

This routine executes the simulator's `step` command. The effect is that the simulator will stop at the next statement of the process that made the call. (Hence, this acts more or less as a permanent breakpoint.)

```
procedure print(...)
```

This procedure can be passed any number of arguments. It prints its argument values on the standard output, followed by a newline. E.g.,

```
x := 5; y := red;
print("x and y are", x, y);
```

results in

```
/q[0]/r> x and y are 5 red
```

where process instance `/q[0]/r` executed the `print` call.

```
procedure show(...)
```

This procedure prints its arguments with their values:

```
show(x+1, y)
```

prints

```
/q[0]/r> example.chp[7:3]
        x + 1 = 6
        y = red
```

Note that it also indicates the location of the call.

```
procedure warning(...)
procedure error(...)
```

These cause a warning and error, respectively, with the argument values as message (in the same way as `print()`).

```
procedure assert(cond: bool)
```

Causes an error if the condition is `false`.

There is also a built-in type `string`, which is an array of `{0..127}`, i.e., an array of ascii values. However, since no array bounds are specified, you cannot index variables of this type. It is useful for some debugging routines.

2.4 Standard I/O

There is a standard module `stdio.chp` that defines procedures for reading and writing files. This module must be imported with `requires` to use these procedures. Obviously, this module is intended for testing and debugging only.

```
type file
const stdin: file
const stdout: file
type file_err = { ok, eof, no_int };
```

Opening and closing of files is done with

```
procedure fopen(res f: file; name: string; mode: string)
procedure fclose(f: file)
```

The `mode` argument is the same as for the `fopen()` function in C. In particular, `"r"` opens the file for reading, `"w"` opens the file for writing (erasing the existing contents).

Reading is done with

```
procedure read_byte(f: file; res x: {0..255}; res err: file_err)
procedure read_int(f: file; res x: int; res err: file_err)
```

`read_byte` reads a single byte (character). If successful it sets `err` to `ok`; otherwise it sets `err` to `eof` and `x` to 0.

`read_int` skips white space, then reads an integer written in the standard CHP notation (Section 1.8.2). The integer may be immediately preceded by a single minus sign. If no digit is found, `err` is set to `no_int` (and `x` to 0). At the end of the file, `err` is set to `eof`.

Writing is done with

```
procedure write_byte(f: file; x: {0..255})
procedure write_string(f: file; x: string)
procedure write_int(f: file; x: int; base: {2..36})
procedure write(f: file; ...)
```

`write_byte` writes a single byte/character. `write_string` writes a sequence of characters, stopping if a 0 character is encountered (the 0 is optional).

`write_int` writes an integer in the specified base. Decimal integers are written without base, hexadecimal is written with `0x`, and all others are written with the *base#* notation.

Finally, `write` writes its arguments in the same way as the `print` procedure of Section 2.3, but without the instance name.